Python Cost Model

Operators in Erik's notes, minus what was covered in lecture:

1. list operations:
   (a) `L1.extend(L2)`
   (b) `L2 = L1[i:j]`
   (c) `b = (x in L)` or `L.index(x)` or `L.find(x)`
2. `tuple` and `str`
3. `set`
4. `heapq`

Other points of interest:

1. equality checking (e.g. `list1 == list2`)
2. lists versus generators

Reference `http://6.006.scripts.mit.edu/~6.006/spring08/wiki/index.php?title=Python_Cost_Model`. The Web site has runtime interpolation for various Python operations. The running times for various-sized inputs were measured, and then a least-square fit was used to find the coefficient for the highest order term in the running time.

Difference between generators and lists. A good explanation is here: `http://wiki.python.org/moin/Generators`

Document Distance

`docdist1.py` is a straightforward solution to the document distance problem, and `docdist{2-8}.py` show algorithmic optimizations that improve the running time.

We start out by understanding the structure of the straightforward implementation, and then we'll look at the changes in each of the successive versions.

docdist1

We first look at `main` to get a high-level view of what's going on in the program.

```
def main():
  if len(sys.argv) != 3:
    print "Usage: docdist1.py filename_1 filename_2"
  else:
    filename_1 = sys.argv[1]
    filename_2 = sys.argv[2]
    sorted_word_list_1 = word_frequencies_for_file(filename_1)
    sorted_word_list_2 = word_frequencies_for_file(filename_2)
    distance = vector_angle(sorted_word_list_1,sorted_word_list_2)
    print "The distance between the documents is: %0.6f (radians)"%distance
```

The method processes the command-line arguments, and calls `word_frequencies_for_file` for each document, then calls `vector_angle` on the resulting lists. How do these methods match up with the three operations outlined in lecture? It seems like `word_frequencies_for_file` is responsible for operation 1 (split each document into words) and operation 2 (count word frequencies), and then `vector_angle` is responsible for operation 3 (compute dot product).

Next up, we'll take a look at `word_frequencies_for_file`.

```
def word_frequencies_for_file(filename):
  line_list = read_file(filename)
  word_list = get_words_from_line_list(line_list)
  freq_mapping = count_frequency(word_list)
  return freq_mapping
```

The method first calls `read_file`, which returns a list of lines in the input file. We'll omit the method code, because it is not particularly interesting, and we'll assume that `read_file`'s running time is proportional to the size of the input file. The input from `read_line` is given to `get_words_from_line_list`, which computes operation 1 (split each document into words). After that, `count_frequency` turns the list of words into a document vector (operation 2).

```
def get_words_from_line_list(L):
  word_list = []
  for line in L:
    words_in_line = get_words_from_string(line)
    word_list = word_list + words_in_line
  return word_list

def get_words_from_string(line):
  word_list = []
  character_list = []
  for c in line:
    if c.isalnum():
      character_list.append(c)
    elif len(character_list)>0:
      word = "".join(character_list)
      word = word.lower()
      word_list.append(word)
      character_list = []
  if len(character_list)>0:
    word = "".join(character_list)
    word = word.lower()
    word_list.append(word)
  return word_list
```

`get_words_from_string` takes one line in the input file and breaks it up into a list of words. TODO: line-by-line analysis. The running time is $O(k)$, where k is the length of the line.

`get_words_from_line_list` calls `get_words_from_string` for each line and combines the lists into one big list. Line 5 looks innocent but is a big performance killer, because using $+$ to combine $\frac{W}{k}$ lists of length k is $O(\frac{W^2}{k})$.

The output of `get_words_from_line_list` is a list of words, like `['a', 'cat', 'in', 'a', 'bag']`. `word_frequencies_from_file` passes this output to `count_frequency`, which turns it into a document vector that counts the number of occurrences of each word, and looks like `[['a', 2], ['cat', 1], ['in', 1], ['bag', 1]]`.

```
def count_frequency(word_list):
  L = []
  for new_word in word_list:
    for entry in L:
      if new_word == entry[0]:
        entry[1] = entry[1] + 1
        break
    else:
      L.append([new_word,1])
  return L
```

The implementation above builds the document vector by takes each word in the input list and looking it up in the list representing the under-construction document vector. In the worst case of a document with all different words, this takes $O(W^2 \times l)$ time, where W is the number of words in the document, and l is the average word length.

`count_frequency` is the last function call in `word_frequencies_for_file`. Next up, `main` calls `vector_angle`, which performs operation 3, computing the distance metric.

```
def vector_angle(L1,L2):
  numerator = inner_product(L1,L2)
  denominator = math.sqrt(inner_product(L1,L1)*inner_product(L2,L2))
  return math.acos(numerator/denominator)
```

The method is a somewhat straightforward implementation of the distance metric

$$\arccos\left(\frac{L1 \cdot L2}{|L1||L2|}\right) = \arccos\left(\frac{L1 \cdot L2}{\sqrt{(L1 \cdot L1)(L2 \cdot L2)}}\right),$$

and delegates to `inner_product` for the hard work of computing cross products.

```
1 def inner_product(L1,L2):
2   sum = 0.0
3   for word1, count1 in L1:
4     for word2, count2 in L2:
5       if word1 == word2:
6         sum += count1 * count2
7   return sum
```

`inner_product` is a straightforward inner-product implementation that checks each each word in the first list against the entire second list. The nested loops at lines 3 and 4 give the algorithm its running time of $\Theta(L_1 L_2)$, where L_1 and L_2 are the lengths of the documents' vectors (the number of unique words in each document).

docdist1 Performance Scorecard

Method	Time
`get_words_from_line_list`	$O(\frac{W^2}{k}) = O(W^2)$
`count_frequency`	$O(WL)$
`word_frequencies_for_file`	$O(W^2)$
`inner_product`	$O(L_1 L_2)$
`vector_angle`	$O(L_1 L_2 + L_1^2 + L_2^2) = O(L_1^2 + L_2^2)$
`main`	$O(W_1^2 + W_2^2)$

We assume that k (number of words per line) is a constant, because the documents will need to fit on screens or paper sheets with a finite length. W (the number of words in a document) is $\geq L$ (the number of unique words in a document). $L_1^2 + L_2^2 + L_1 L_2 = O(L_1^2 + L_2^2)$ because $L_1^2 + L_2^2 \geq L_1 L_2$. Proof (assuming $L_1, L_2 \geq 0$):

$$\begin{aligned}
(L_1 - L_2)^2 &\geq 0 \\
L_1^2 + L_2^2 - 2L_1 L_2 &\geq 0 \\
L_1^2 + L_2^2 &\geq 2L_1 L_2 \\
L_1^2 + L_2^2 &\geq L_1 L_2
\end{aligned}$$

docdist2

The document distance code invokes the Python profiler to identify the code that takes up the most CPU time. This ensures that we get the biggest returns on our optimization efforts.

```
if __name__ == "__main__":
  import cProfile
  cProfile.run("main()")
```

You can profile existing programs without changing them by adding `-m cProfile -s Time` to Python's command line. For example, the command below will run and profile `program.py`.

```
python -m cProfile -s time program.py
```

The profiler output for docdist1 shows that the biggest time drain is `get_words_from_line_list`. The problem is that when the + operator is used to concatenate lists, it needs to create a new list and copy the elements of both its operands. Replacing + with `extend` yields a 30% runtime improvement.

```
def get_words_from_line_list(L):
  word_list = []
  for line in L:
    words_in_line = get_words_from_string(line)
    word_list.extend(words_in_line)
  return word_list
```

`extend` adds all the elements of an m-element list to an n-element list in $\Theta(1+m)$, as opposed to +, which needs to create a new list, and therefore takes $\Theta(1+n+m)$ time. So concatenating $\frac{W}{k}$ lists of k elements takes $\sum^{\frac{W}{k}} k = \Theta(W)$ time.

docdist2 Performance Scorecard

Method	Time
`get_words_from_line_list`	$O(W)$
`count_frequency`	$O(WL)$
`word_frequencies_for_file`	$O(WL)$
`inner_product`	$O(L_1 L_2)$
`vector_angle`	$O(L_1^2 + L_2^2)$
`main`	$O(W_1 L_1 + W_2 L_2)$

docdist3

Profiling docdist2 points to `count_frequency` and `inner_product` as good targets for optimizations. We'll speed up `inner_product` by switching to a fast algorithm. However, the algorithm assumes that the words in the document vectors are sorted. For example, `[['a', 2], ['cat', 1], ['in', 1], ['bag', 1]]` needs to become `[['a', 2], ['bag', 1], ['cat', 1], ['in', 1]]`.

First off, we add a step to `word_frequencies_for_file` that sorts the document vector produced by `count_frequency`.

```
def word_frequencies_for_file(filename):
    line_list = read_file(filename)
    word_list = get_words_from_line_list(line_list)
    freq_mapping = count_frequency(word_list)
    insertion_sort(freq_mapping)
    return freq_mapping
```

Then we implement `insertion_sort` using the algorithm in the textbook.

```
def insertion_sort(A):
  for j in range(len(A)):
    key = A[j]
    i = j-1
    while i>-1 and A[i]>key:
        A[i+1] = A[i]
        i = i-1
    A[i+1] = key
  return A
```

Insertion sort runs in $O(L^2)$ time, where L is the length of the document vector to be sorted.

Finally, `inner_product` is re-implemented using a similar algorithm to the merging step in Merge Sort.

```
def inner_product(L1,L2):
  sum = 0.0
  i = 0
  j = 0
  while i<len(L1) and j<len(L2):
    # L1[i:] and L2[j:] yet to be processed
    if L1[i][0] == L2[j][0]:
      # both vectors have this word
      sum += L1[i][1] * L2[j][1]
      i += 1
      j += 1
    elif L1[i][0] < L2[j][0]:
      # word L1[i][0] is in L1 but not L2
      i += 1
    else:
      # word L2[j][0] is in L2 but not L1
      j += 1
  return sum
```

The new implementation runs in $\Theta(L_1 + L_2)$, where L_1 and L_2 are the lengths of the two document vectors. We observe that the running time for `inner_product` (and therefore for `vector_angle`) is asymptotically optimal, because any algorithm that computes the inner product will have to read the two vectors, and reading will take $\Omega(L_1 + L_2)$ time.

docdist3 Performance Scorecard

Method	Time
`get_words_from_line_list`	$O(W)$
`count_frequency`	$O(WL)$
`insertion_sort`	$O(L^2)$
`word_frequencies_for_file`	$O(WL + L^2) = O(WL)$
`inner_product`	$O(L_1 + L_2)$
`vector_angle`	$O(L_1 + L_2)$
`main`	$O(W_1L_1 + W_2L_2)$

docdist4

The next iteration addresses `count_frequency`, which is the biggest time consumer at the moment.

```
def count_frequency(word_list):
  D = {}
  for new_word in word_list:
    if new_word in D:
      D[new_word] = D[new_word]+1
    else:
      D[new_word] = 1
  return D.items()
```

The new implementation uses Python dictionaries. The dictionaries are implemented using hash tables, which will be presented in a future lecture. The salient feature of hash tables is that inserting an element using `dictionary[key] = value` and looking up an element using `dictionary[key]` both run in $O(1)$ expected time.

Instead of storing the document vector under construction in a list, the new implementation uses a dictionary. The keys are the words in the document, and the value are the number of times each word appears in the document. Since both insertion (line 5) and lookup (line 7) take $O(1)$ time, building a document vector out of W words takes $O(W)$ time.

docdist4 Performance Scorecard

Method	Time
`get_words_from_line_list`	$O(W)$
`count_frequency`	$O(W)$
`insertion_sort`	$O(L^2)$
`word_frequencies_for_file`	$O(W + L^2) = O(L^2)$
`inner_product`	$O(L_1 + L_2)$
`vector_angle`	$O(L_1 + L_2)$
`main`	$O(L_1^2 + L_2^2)$

docdist5

This iteration simplifies `get_words_from_string` that breaks up lines into words. First off, the standard library function `string.translate` is used for converting uppercase characters to lowercase, and for converting punctuation to spaces. Second, the `split` method on strings is used to break up a line into words.

```
translation_table = string.maketrans(string.punctuation+string.uppercase,
                   " "*len(string.punctuation)+string.lowercase)

def get_words_from_string(line):
  line = line.translate(translation_table)
  word_list = line.split()
  return word_list
```

The main benefit of this change is that 23 lines of code are replaced with 5 lines of code. This makes the implementation easier to analyze. A side benefit is that many functions in the Python standard library are implemented in directly in C (a low-level programming language that is very close to machine code), which gives them better performance. Although the running time is asymptotically the same, the hidden constants are much better for the C code than for our Python code presented in docdist1.

docdist5 Performance Scorecard

Identical to the docdist4 scorecard.

docdist6

Now that all the distractions are out of the way, it's time to tackle insertion_sort, which is takes up the most CPU time, by far, in the profiler output for docdist5.

The advantages of insertion sort are that it sorts in place, and it is simple to implement. However, its worst-case running time is $O(N^2)$ for an N-element array. We'll replace insertion sort with a better algorithm, merge sort. Merge sort is not in place, so we'll need to modify word_frequencies_for_file.

```
def word_frequencies_for_file(filename):
  line_list = read_file(filename)
  word_list = get_words_from_line_list(line_list)
  freq_mapping = count_frequency(word_list)
  freq_mapping = merge_sort(freq_mapping)
  return freq_mapping
```

The merge_sort implementation closely follows the pseudocode in the textbook.

```
def merge_sort(A):
  n = len(A)
  if n==1:
    return A
  mid = n//2
  L = merge_sort(A[:mid])
  R = merge_sort(A[mid:])
  return merge(L,R)

def merge(L,R):
  i = 0
  j = 0
  answer = []
  while i<len(L) and j<len(R):
    if L[i]<R[j]:
      answer.append(L[i])
      i += 1
    else:
      answer.append(R[j])
      j += 1
  if i<len(L):
    answer.extend(L[i:])
  if j<len(R):
    answer.extend(R[j:])
  return answer
```

The textbook proves that merge_sort runs in $\Theta(n \log n)$ time. You should apply your knowledge of the Python cost model to convince yourself that the implementation above also runs in $\Theta(n \log n)$ time.

docdist6 Performance Scorecard

Method	Time
`get_words_from_line_list`	$O(W)$
`count_frequency`	$O(W)$
`merge_sort`	$O(L \log L)$
`word_frequencies_for_file`	$O(W + L \log L) = O(L \log L)$
`inner_product`	$O(L_1 + L_2)$
`vector_angle`	$O(L_1 + L_2)$
`main`	$O(L_1 \log L_1 + L_2 \log L_2)$

docdist7

Switching to merge sort improved the running time dramatically. However, if we look at docdist6's profiler output, we notice that `merge` is the function with the biggest runtime footprint. Merge sort's performance in practice is great, so it seems that the only way to make our code faster is to get rid of sorting altogether.

This iteration switches away from the sorted list representation of document vectors, and instead uses the Python dictionary representation that was introduced in docdist4. `count_frequency` already used that representation internally, so we only need to remove the code that converted the Python dictionary to a list.

```python
def count_frequency(word_list):
  D = {}
  for new_word in word_list:
    if new_word in D:
      D[new_word] = D[new_word]+1
    else:
      D[new_word] = 1
  return D
```

This method still takes $O(W)$ time to process a W-word document.

`word_frequencies_for_file` does not call `merge_sort` anymore, and instead returns the dictionary built by `count_frequency`.

```python
def word_frequencies_for_file(filename):
    line_list = read_file(filename)
    word_list = get_words_from_line_list(line_list)
    freq_mapping = count_frequency(word_list)
    return freq_mapping
```

Next up, `inner_product` makes uses dictionary lookups instead of merging sorted lists.

```python
def inner_product(D1,D2):
  sum = 0.0
  for key in D1:
    if key in D2:
      sum += D1[key] * D2[key]
  return sum
```

The logic is quite similar to the straightforward `inner_product` in docdist1. Each word in the first document vector is looked up in the second document vector. However, because the document vectors are dictionaries, each takes $O(1)$ time, and `inner_product` runs in $O(L_1)$ time, where L_1 is the length of the first document's vector.

docdist7 Performance Scorecard

Method	Time
`get_words_from_line_list`	$O(W)$
`count_frequency`	$O(W)$
`word_frequencies_for_file`	$O(W)$
`inner_product`	$O(L_1 + L_2)$
`vector_angle`	$O(L_1 + L_2)$
`main`	$O(W_1 + W_2)$

docdist8

At this point, all the algorithms in our solution are asymptotically optimal. We can easily prove this, by noting that each of the 3 main operations runs in time proportional to its input size, and each operation needs to read all its input to produce its output. However, there is still room for optimizing and simplifying the code.

There is no reason to read each document line by line, and then break up each line into words. The last iteration processes reads each document into one large string, and breaks up the entire document into words at once.

First off, `read_file` is modified to return a single string, instead of an array of strings. Then, `get_words_from_line_list` is renamed to `get_words_from_file`, and is simplified, because it doesn't need to iterate over a list of lines anymore. Last, `word_frequencies_for_file` is updated to reflect the renaming.

```
def get_words_from_text(text):
    text = text.translate(translation_table)
    word_list = text.split()
    return word_list

def word_frequencies_for_file(filename):
    text = read_file(filename)
    word_list = get_words_from_text(text)
    freq_mapping = count_frequency(word_list)
    return freq_mapping
```

docdist8 Performance Scorecard

Method	Time
`get_words_from_text`	$O(W)$
`count_frequency`	$O(W)$
`word_frequencies_for_file`	$O(W)$
`inner_product`	$O(L_1 + L_2)$
`vector_angle`	$O(L_1 + L_2)$
`main`	$O(W_1 + W_2)$